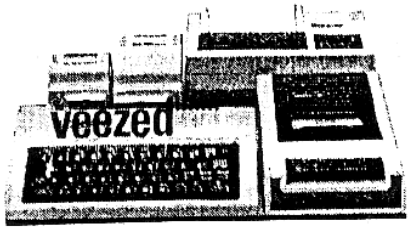


COMPATIBILITY

Asterisks in DSE advertisements suggested that the VZ-200 16K Expansion Module is compatible with the VZ-300. Unfortunately, this is not the case. However, whereas the 200 had only 6K of in-buit memory, the 300 has 16K, so maybe you will never need an Expansion Unit for your VZ-300.

But why won't a VZ-200 16K Unit suit the VZ-300? The 300's 16K extends to memory address 63487 whilst the 200's module begins at 55248. The only part of the unit usable is the part between 55248 and 63487, about a mere 1K. There is a feature in one of the magazines describing how to perform brain surgery on the VZ-200 module to make it suitable for the VZ-300, but you will need more than a hammer and chisel.

Now if you plug a VZ-300 Expansion Unit into a VZ-200, and I cannot imagine why you should, but if you did there would be a gap of 10K in the memory addresses. The computer would ignore the module altogether.

In the case of the 300's 64K module, this too presents a problem. The VZ-300 can accomodate a total of only 34K. A system known as Bank Switching puts 3 banks of 16K in parallel with another 16K available for immediate use. The three banks all have the same addresses and a program command selects which of them is to be used. The command cannot be in BASIC, it must be Machine Code.

~~Should you consider buying a Disk Drive for either the 200 or the 300, although it does not use any of the free RAM, it does plug into the expansion socket preventing any increase from an expansion module. This would leave the VZ-200 with only its 6K.~~ DELETED.
The advantage of Disks over cassettes is that they will LOAD & SAVE 14 times faster. A Disk has a storage capacity of about 78K. The Disk Commands are already built into VeeZeds.

What a pity the designers of the VeeZeds blocked off in-built commands, such as ONGOTO/GOSUB, DEFINT, VARPTR, FRE etc. With all these extra commands the VZ-300 with a Disk Drive would be superior than other computers costing many times the price.

In Studio Ad Lib we aim to show you how to access these extra commands very easily.

TRON = Trace on. TROFF = Trace off.

The best way of describing TRON and TROFF is to demonstrate...

ENTER and RUN this...

```
10 CLS:PRINT@320,"MICRO MAGIC"
20 FORI=1TO10:SOUND16,3;18,6:NEXT
30 GOTO30
```

Now ADD 5 POKE31003,175 and RUN again. What you see is your line numbers being printed as they are being RUN. It is up to you how you use TRON when debugging a program. Obviously, the closer to the BUG the better. To turn the trace off enter POKE31003,0 (TROFF).

DEFINT, DEFSNG, DEFDBL and DEFSTR.

In your VeeZed is a VARIABLE DECLARATION TABLE.
It is located between 30977 and 31002. This VDT is 26 bytes - one for each letter of the alphabet. For reasons best known to the computer's designers, there is no direct access from the keyboard.

On switch-on the whole of the VDT is initialised to SINGLE PRECISION VARIABLES. This means that there is a 4 in every location. However, it is possible to change the 4s to other numbers in order to create other kinds of variables. Using A as an example...

2 = INTEGER 3 = STRING 4 = SINGLE PRECISION 8 = DOUBLE PRECISION

An INTEGER VARIABLE will accept WHOLE NUMBERS and entered as A%.

A STRING VARIABLE will accept up to 255 characters and entered as A$.

A SINGLE PRECISION VARIABLE will accept decimal values of numbers to 6 significant places and entered as A.

A DOUBLE PRECISION VARIABLE will accept decimal numbers of 14 to 16 significant places and entered as A# (Not accepted by the VeeZed).

To change the VeeZed VDT to, say, make every letter (A to Z) into INTEGER VARIABLES, equivilent to DEFINT A-Z, the following will do it.

```
10 FORI=30977TO31002:POKEI,2:NEXT
```

The following can be used to define an individual variable...

```
10 POKE30912+ASC("A"),3
```

which would define A as a STRING VARIABLE (DEFSTRA). Whenever A is used it will be regarded as a STRING VARIABLE without the dollar sign. Once a letter has been defined there is no longer need for % or $.

FINDING YOURSELF IN MEMORY

```
 10 PRINT"ALPHABET"
 20 FORS=31410TO31510
 30 C$=CHR$(PEEK((S))
 40 IFC$<>"A"THEN80
 50 D$=CHR$(PEEK(S+1))
 60 IFD$<>"L"THEN80
 70 PRINTS
 80 NEXT
 100 INPUT"STARTING LOCATION";X
 110 FORZ=XTOX+7
 120 PRINTZ;CHR$(PEEK(Z))
 130 NEXT
RUN
ALPHABET                 you
 31471                   enter
STARTING LOCATION?       31471
 31471 A
 31472 L
 31473 P
 31474 H
 31475 A
 31476 B
 31477 E
 31478 T
READY
```

Now that we know WHERE the first line of the program is stored, and we know WHAT the TOKEN for PRINT is, making use of this knowledge could be a first step into Machine Code programming.

TOKENS

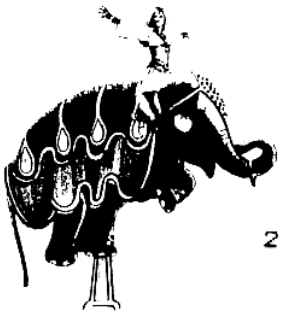
Line 10 is DUMMY TEXT.
Line 20 contains Memory Locations.
Lines 30 and 40 allows the LOOP to run until the letter A is found.
Lines 50 and 60 checks the next locations for the letter L.
If both an A and an L are found then the memory location of A is printed. It may not be the word ALPHABET in which case the search will have to be restarted using other memory locations in the loop. In this example the locations are correct and the following is what you should have....

Instructions, such as PRINT, are coded. The code is called a TOKEN. The TOKEN for PRINT in the above experiment must be just before 31471. Agree? So, RUN again, but this time enter 31467 and you will see that THAT location contains ASCII 178 (SHIFT 2 ■). This is the TOKEN for PRINT.

CONSIDER THIS...

```
    10 PRINT"ELEPHANT"
    20 FORA=31469TO31479
    30 READB:POKEA,B
    40 NEXT
    50 DATA178,34,67,79,77,80,85,84,69,82,34
    RUN
    ELEPHANT
    READY
    RUN
    COMPUTER
    READY
    LIST10
    10 PRINT"COMPUTER"
```

Whatever happened to the ELEPHANT???



TOKENS			1st line address: 31469				
128	END	160	OUT	192	VARPTR	224	EXP
129	FOR	161	ON	193	USR	225	COS
130	RESET	162		194	ERL	226	SIN
131	SET	163		195	ERR	227	TAN
132	CLS	164		196	STRING$	228	ATN
133		165		197		229	PEEK
134	RANDOM	166		198	POINT	230	
135	NEXT	167		199		231	
136	DATA	168		200	MEM	232	
137	INPUT	169		201	INKEY$	233	
138	DIM	170		202	THEN	234	
139	READ	171		203	NOT	235	
140	LET	172		204	STEP	236	
141	GOTO	173		205	+	237	
142	RUN	174	(RESET)	206	-	238	
143	IF	175	LPRINT	207	*	239	CINT
144	RESTORE	176		208	/	240	CSNG
145	GOSUB	177	POKE	209	^	241	CDBL
146	RETURN	178	PRINT	210	AND	242	FIX
147	REM	179	CONT	211	OR	243	LEN
148	STOP	180	LIST	212	>	244	STR$
149	(E)LSE	181	LLIST	213	=	245	VAL
150	COPY	182	DELETE	214	<	246	ASC
151	COLOR	183	AUTO	215	SGN	247	CHR$
152	VERIFY	184	CLEAR	216	INT	248	LEFT$
153	DEFINT	185	CLOAD	217	ABS	249	RIGHT$
154	DEFSNG	186	CSAVE	218	FRE	250	MID$
155	DEFDBL	187	NEW	219	INP	251	
156	CRUN	188	TAB(	220	POS	252	
157	MODE	189	TO	221	SQR	253	
158	ERROR	190		222	RND	254	
159	RESUME	191	USING	223	LOG	255	

NOTE: TOKEN 149 = ELSE (but the E is missing).
TOKEN 158 = ERROR (but can return SOUND).
TOKEN 174 = RESET (like switching off and on).
TOKEN 196 = STRING$ (but not the same as STR$).
Blank lines could be DISK COMMANDS but not identified.

FOR, TO, NEXT, RETURN, THEN, Etc., Etc., are COMMANDS (sometimes called FUNCTIONS. They tell the computer to DO something. Every command has a TOKEN, which is similar to ASCII numbers, except that the computer reads them as commands. FINDING YOURSELF IN MEMORY, I thought, would help you to understand what TOKENS were all about. In the List of Tokens above, those that are not accessable from the keyboard are underlined. They are easily used. We simply have to POKE them into our programs. These commands are built-in to the compluter just as PRINT and LIST and all those others on the keyboard. The difference is that their names, such as AUTO and DELETE are not in the computers vocabulary.

Let MICRO MAGIC show you just how *EASY* it is to use them.

In the FIRST line of any BASIC program, the FIRST memory location AFTER the Line Number is **31469.**

This is regardless of what the line number is. All line numbers take up the same amount of memory. The very next memory location (address) is always 31469. If the first command after the first line numbers was REM, and we typed POKE31469,132 the first line would be changed from REM to CLS. If, using the Tokens Table above, we POKEd 183, because the VeeZed does not have AUTO in its vocabluary, the first line would appear empty when LISTed. The TOKEN (183), however, is firmly in place, RUNning the line will initiate Automatic Line Numbering.

VZ
Invisible Commands

Suppose you wanted to use MEM. This is the command that gives the amount of unused memory. Following are three methods of putting the TOKEN for MEM into a program. As MEM must be preceded by the command PRINT (in address 31469), then the TOKEN for MEM must need to be in the next address, 31470. This is the address we reserve with a *dummy character*, we use a HASH sign #.

FIRST LINE METHOD

10 PRINT#*RETURN*
POKE31470,200*RETURN*
When LISTed line 10 appears as **10 PRINT** but the MEM TOKEN is in place and when RUN will give the unused memory space.

GOSUB METHOD

MEM in the first line might be inconvenient. This method lets you use MEM anywhere in a program. The line receiving the token MUST be entered first.
20 PRINT#*RETURN*
POKE31470,200*RETURN*
Note: the line number has been made 20. When this is entered type the following two lines.
10 GOTO100*RETURN*
30 RETURN*RETURN*
Line 100 now becomes the true beginning of your program and GOSUB20 can be used anywhere. Test by using the following...
100 CLS:GOSUB20*RETURN*
110 GOTO110*RETURN*

PEEK ADDRESS METHOD

To put MEM anywhere in a program requires determining the EXACT address. When a program is being entered, at the spot where MEM is required, enter the dummy character (#). The program must be correct up to this point before proceeding. Let's assume the last line of the incompleted program is...
130 CLS:PRINT"MEMORY";:#
Enter directly the following:
PRINTPEEK(30969)+256*PEEK(30970)-4
The number that is printed is the memory address of the #, which is where the TOKEN for MEM is to go. POKE the number with 200 (for MEM). If the number is greater than 32767, SUBTRACT 65536. Continue with entering the program from the next program line, (line 140 in this example). Care must be taken not to overwrite the invisible command.

RANDOM	10 #	POKE31469,134
	Reseeds RND(x) to give different series of random numbers when routine is run.	
DEFINT	10 #A,B	POKE31469,153
	Defines all variables beginning with A and B as INTEGERS.	
DEFSNG	10 #C,D	POKE31469,154
	Defines all variables beginning with C and D as SINGLE PRECISION. (6/7 floating point).	
DEFDBL	10 #E,F	POKE31469,155
	Defines all variables beginning with E and F as DOUBLE PRECISION. (16/17 floating point).	
ON	10 #	POKE31469,161
	Used with ONGOTO, ONGOSUB and ONERROR. (see below).	
ERROR	10 #*	POKE31469,161:POKE31470,158
	Used for ON ERROR plus GOTO/GOSUB (Line number).	
RESUME	10 #	POKE31469,159
	After ERROR return to ERROR point.	
	10 #100	
	After ERROR GOTO100.	
	10 #100NEXT	
	After ERROR return to the line after ERROR point.	
DELETE	10 #120-200	POKE31469,182
	DELETE lines 120 to 200 inclusive.	
AUTO	10 #	POKE31469,183:RUN
	Automatically prints line numbers starting at 10, increments of 10.	
	10 #100,10	
	Automatically prints lines numbers beginning at 100, increments of 10.	
	NOTE: If AUTO halted by BREAK POKE30945,175 will restart AUTO from where it was halted.	
VARPTR	10 #(X)	POKE31469,192
	Used to locate the Memory Address of a variable.	
STRING$	10 PRINT#(12,45)	POKE31470,196
	Will PRINT 12 Asterisks (*). Maximum length of string is 256 characters.	
MEM	10 PRINT#	POKE31470,200
	Returns the amount of unused memory remaining.	
FRE	10 PRINT#(A)	POKE31470,218
	Returns the number of unused BYTES remaining.	
FRE	10 PRINT#(A$)	POKE31470,218
	Returns the number of unused BYTES remaining in the reserved string spaces.	
CINT	10 #Z	POKE31469,239
	Removes all digits after the decimal point.	
CSNG	10 #Z	POKE31469,240
	Converts numeric variables from DOUBLE to SINGLE PRECISION.	
CDBL	10 #Z	POKE31469,241
	Converts numeric variables from SINGLE to DOUBLE PRECISION.	
FIX	10 A=#(n)	POKE31471,242
	Removes all digits to the RIGHT of the decimal point (does not ROUND-OFF negative numbers).	
ERL	10 PRINT#	POKE31470,194
	Returns the line number from which a program branched to ERROR routine.	
ERR	10 PRINT#	POKE31470,195
	Returns a value related to the type of error.	
POS	10 A=#(X)	POKE31471,220
	Returns the current CURSOR position. (x) is a dummy argument.	